공학
트렌드

소프트웨어공학센터 경영지원TF팀

대용량 서비스를 지탱하는 분산 캐시 시스템 Part 1: 캐시 시스템의 역사

2015. 2. 10. [122호]

Contents

정보통신산업진흥원 부설
nipa 소프트웨어공학센터

Ⅰ. 분산 캐시(Cache) 시스템의 이해

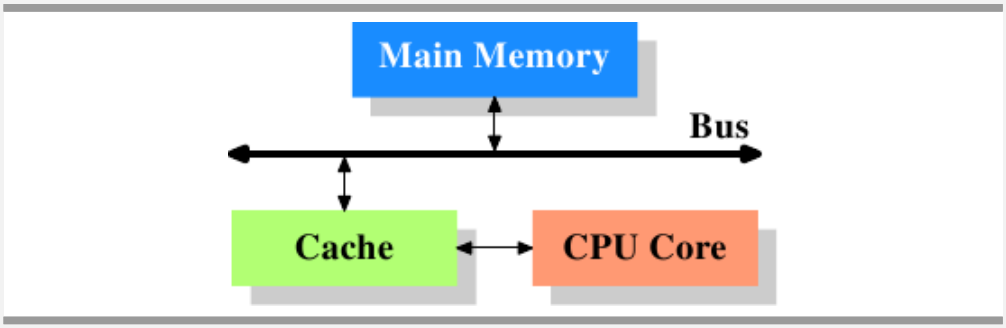
일반적으로 캐시라 하면 데이터나 값을 미리 복사해 놓는 임시 장소를 가리킨다.[1] 계산 또는 저장된 값을 읽기 위해 특정 장소에 접근하는 시간이 오래 걸릴 경우, 해당 소요 시간을 줄이기 위해서 캐시가 만들어졌다. 즉, 캐시는 빠른 접근을 가능하게 하는 효율성에 집중되어 있다.

캐시는 CPU 성능을 높이기 위한 L1, L2, L3 캐시를 사용하는 CPU 캐시, 디스크의 내용을 RAM에 저장하는 DISK 캐시, web 브라우저의 캐시나 iOS, Android와 같은 미들웨어, 애플리케이션에서 사용하는 단말 애플리케이션 단위의 캐시, DB나 웹 서버, 대용량 서버에서 사용하는 분산 캐시 등으로 크게 나눌 수 있을 것이다.

1.1 CPU 캐시를 통해 보는 캐시 구축 관점

CPU 캐시를 잠시 살펴보고 캐시의 개념에 대해 다시 살펴보고자 한다. CPU는 Main Bus를 통해 계산될 또는 계산된 메모리로 접근하는데, 시간의 속도를 줄이기 위해서 Register와 Cache를 두게 되었다. 〈그림 1〉과 같이 CPU 캐시 환경을 개념도로 볼 수 있다. CPU는 Cache에 데이터가 있는지 확인하고 해당 데이터가 존재하지 않으면, Main Memory로 접근한다. Main Memory로 접근을 하지 않을 때는 빠르다. 그러나 빠른 처리를 위해 캐시를 추가하면 비용이 높아진다.

그림 1_CPU 캐시 환경

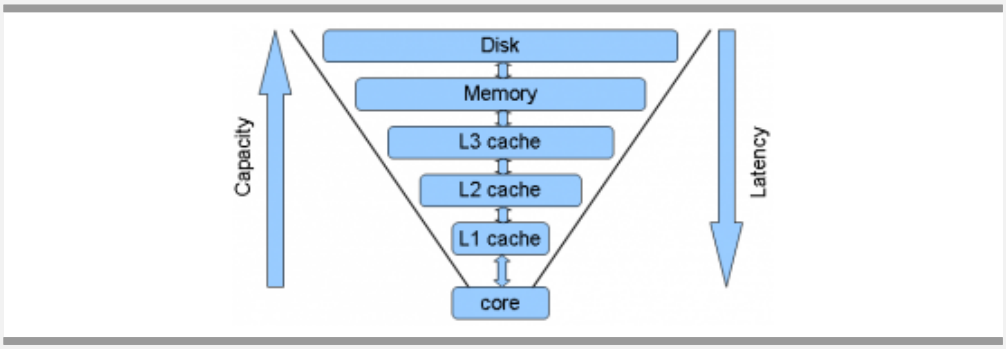


출처 : http://lwn.net/Articles/252125/

1) 출처 : http://ko.wikipedia.org/wiki/%EC%BA%90%EC%8B%9C

〈그림 2〉는 CPU와 내부 자원과의 통신 시 용량과 응답 속도 지연 정보 간의 상관도를 보인다. CPU 캐시를 사용하면 빠를 수는 있으나 사용할 수 있는 예산이 정해져 있으니 용량이 큰 메모리나 디스크를 사용해야 할 수 있다.

그림 2_내부 시스템 간의 용량과 응답 속도 지연 정보



출처 : http://www.buildagamingpc.org/gaming-cpu/cpu-cache

따라서 캐시 시스템을 구축할 때는 CPU 캐시와 비슷하게 비용, 응답 속도, 용량을 잘 고민해야 한다. 분산 캐시 시스템을 구축할 때 이런 고민은 동일하게 적용된다.

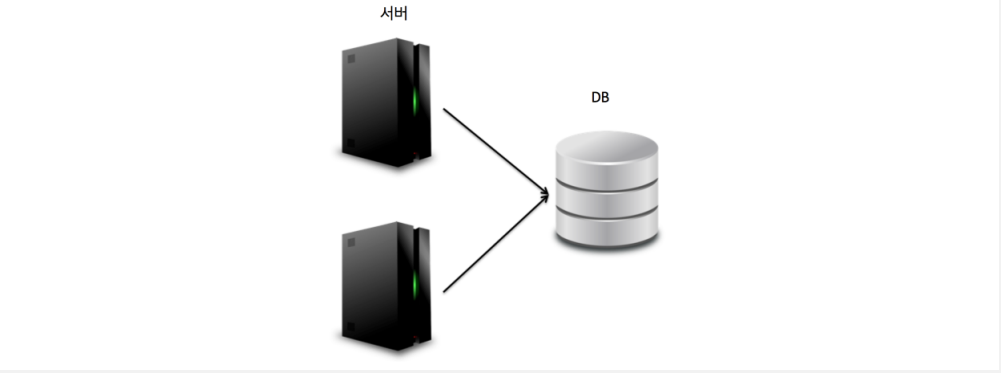
그리고, 또 하나 고민해야 할 요소는 데이터의 무결성과 캐시를 언제 사용할지에 대한 부분이다. 데이터가 변경되었는데, 캐시에 적용되지 않았다면 어떻게 해야 할 지에 대한 고민이 필요하다. 캐시에 저장하는 것을 캐싱(Caching)이라고 불리는데, 어떻게 캐싱할 지 잘 정해야 한다. write가 read보다 더 많이 호출되고 있는 상황에서 캐싱은 큰 의미가 없을 수 있다. read가 write보다 많은 경우에 캐싱이 더 많은 성능을 줄 수 있다.

II. 서버 환경의 캐시와 역사

저자가 분산 환경 또는 서버 환경에서 개발하면서 만들고 경험했던 캐시 시스템을 소개하고, 분산 캐시의 대표 주자인 redis와 memcached 등이 만들어지게 되는 배경을 살펴보고자 한다.

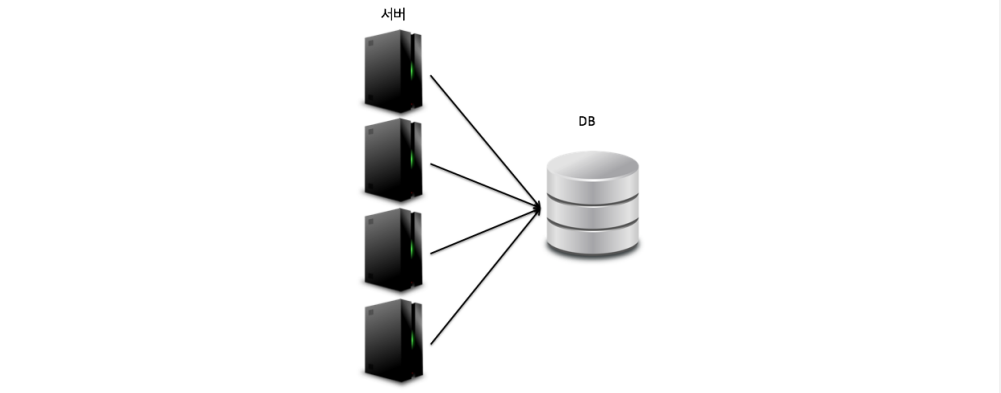
웹 서비스와 같은 일반적이고 기초적인 서버, DB의 아키텍처는 〈그림 3〉과 같다. L4와 같은 네트워 장치와 방화벽과 요청에 대한 내용은 그림에서 제외했다. 서버에서 요청을 받고 처리하고 처리된 정보를 DB에 저장하거나 서버로 데이터를 전달한다.

그림 3_기초적인 서버, DB 연동 아키텍처



트래픽이 많아지면, 서버와 DB를 증설한다. 일단 서버를 증설해 가용한 용량 크기만큼 서버를 증설한다. 〈그림 4〉와 같은 아키텍처와 서버를 증설하는 형태가 된다. 점점 DB는 병목 현상이 발생하기 시작한다. 과거에는 발생하지 않았던 동시성 이슈, 성능 상 이슈가 있는 SQL 쿼리, DB의 데이터양/메모리 증가, 서버에서 DB 간의 connection 비용이 발생하기 시작한다. 그 중에 DB connection은 비용을 많이 증가 시킬 수 있는 아키텍처이다. 상용 DB 일수록 메모리 사용 비용이 크다.

그림 4_트래픽 증가에 따른 서버 증설 아키텍처

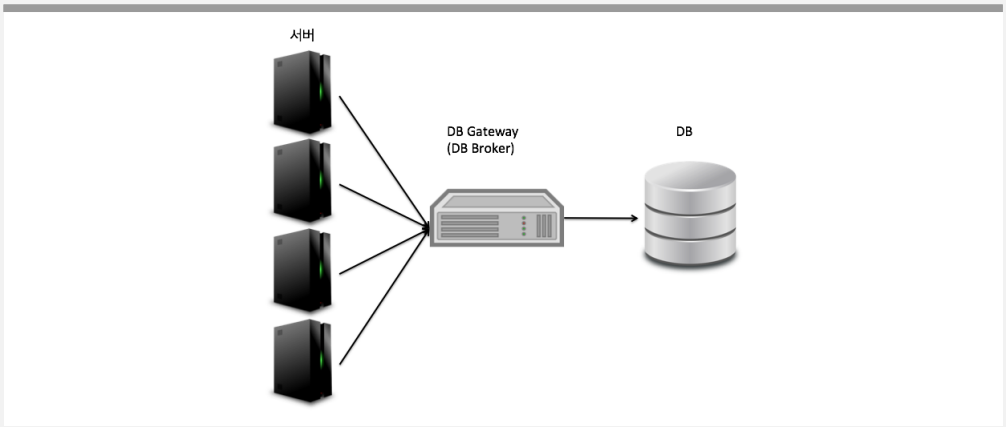


DB를 최적화하는 작업과 SQL 쿼리(Query)를 최적화하는 작업도 동시에 이뤄진다. DB의 버퍼 캐시는 사용자가 저장한 데이터를 파일에 읽으려 할 때 사용하는 캐시 공간인데, DB의 I/O 작업을 최소화 할 수 있다. 그리고 DB의 index를 최적화하고, SQL 쿼리를 prepared statement로 변경하고, 여러 번 할 수 있는 쿼리를 한 번에 요청할 수 있지만 복잡한 query statement로 변경한다. 또한 트랜잭션(transaction)의 여러 레벨을 잘 활용하는 코드를 작성한다.

한편 서버 애플리케이션은 DB로 요청하는 SQL Query를 최적화하는 작업을 진행한다. 서버 애플리케이션에서 DB 간의 connection 에 대한 context switching 비용, 성능 저하를

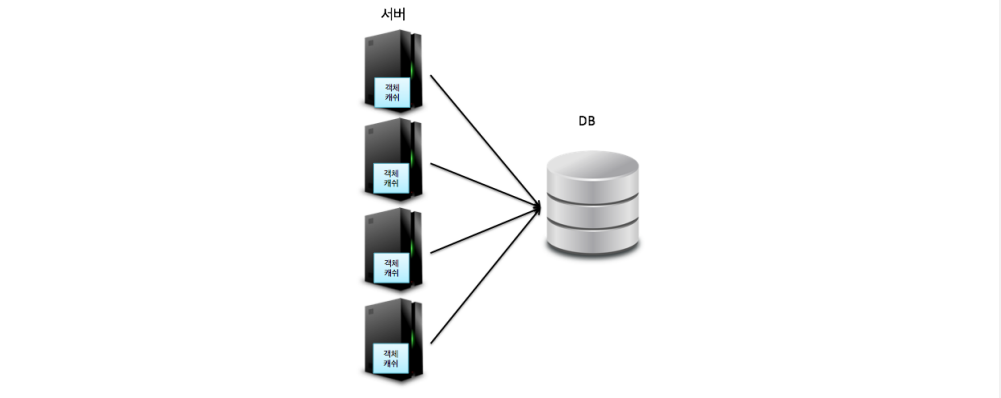
초래하는 코드를 수정하거나, DB connection을 재사용 할 수 있는 connection pool을 구현했다. 그리고 대용량 트래픽이 DB connection pool이 아닌 DB connection을 효율적으로 관리하는 DB Gateway(DB Broker)와 같은 DB 소프트웨어, 상용 미들웨어, 오픈 소스가 나타나기 시작하고 실제로 많이 사용되고 있다.

그림 5_DB의 부하를 경감하는 DB Gateway를 사용한 아키텍처



대부분의 애플리케이션은 서버의 프로세스로 실행되며 각 프로세스의 메모리는 서로 간에 공유할 수 없는 상황이다. 따라서 각 애플리케이션에서 DB로의 요청을 매번 전송하지 않도록 하기 위해 프로세스 내 메모리를 사용한다. 예를 들어, 애플리케이션은 Collection을 사용하여 개수를 제한하는 저장소로 사용하는 방식이다. 필요하다면 LRU[2]를 구현하여 사용한다. 현재까지도 LRU 캐싱은 분산 캐시 시스템 없이 애플리케이션 자체에서 해결할 수 있는 방법이다. 분산 캐시와 혼동이 되지 않기 위해 '객체(Object) 캐시'[3]라 표현한다.

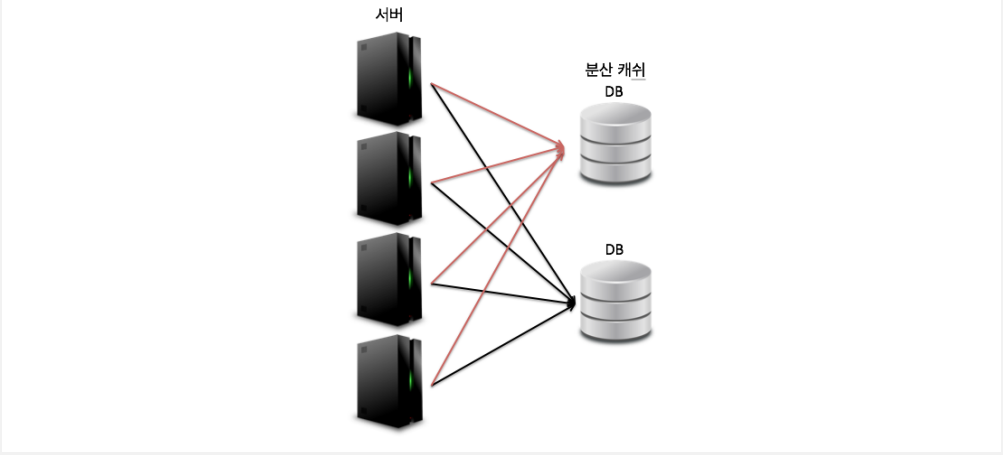
그림 6_서버 애플리케이션 내 객체 캐시를 활용하여 성능 효과와 DB의 부하를 절감하는 아키텍처



2) LRU(Least Recently Used): 최근에 사용하지 않는 캐시를 먼저 삭제하는 방식
3) 저자가 분류를 위해서 사용한 단어로 '객체 캐시'라 표현한다.

객체 캐시는 서버들 간의 공유는 되지 않았지만, 시스템을 따로 구축하지 않고도 Collection을 사용하는 것만으로 성능이 개선된다. 특히 오픈 소스 진영도 가세해서 ehcache나 guava cache와 같은 객체 캐시 라이브러리가 만들어졌다. 단순 저장 방법에서 다양한 알고리즘을 사용할 수 있는 오픈 소스 라이브러리이다. 네트웍이 빨라지고 안정성이 좋아지면서 중복성을 줄이고 더 많은 데이터를 저장할 수 있는 서버나 DB를 두어 분산 캐시 시스템이 나타났다. 〈그림 7〉은 애플리케이션에서 분산 캐시를 DB로 활용한 사례이다. 객체 캐시와 DB를 활용한 '분산 캐시 DB'[4]를 활용했다. 지금은 잘 활용하지 않는데, 해당 DB에는 객체 캐시와 비슷한 Map을 Table에 저장하며 마치 키-값 아키텍처로 바로 사용할 수 있는 형태이다. 분산 캐시 DB는 객체 캐시와 분산 캐시의 중간 형태 정도로 이해하면 될 것이다.

그림 7_분산 캐시 DB를 구축한 아키텍처



느린 DB 접근 응답 속도보다 상대적으로 빠른 응답 속도(O(1))를 가진 분산 캐시 시스템이 나타나기 시작했다. DB를 가볍게 쓸 수 있고, DBA가 꼭 필요한 DB 보다 개발자가 편리하게 다루기 쉽고, 쉽게 이해할 수 있는 Map 개념을 가진 키-값(key-value)을 선호했다. Devops 열풍도 함께 불면서 기존의 문제점을 해결하면서 높은 수준의 성능, 확장성, 유지보수성, 빠른 배포 등을 특징으로 하는 분산 캐시 시스템을 필요로 하게 되었다. 〈그림 8〉은 Redis 또는 Memcached와 같은 분산 캐시 시스템을 사용한 아키텍처라고 할 수 있다. (분산 캐시는 언급한 Redis, Memcached 말고 Terracotta cache, Jboss cache 등이 있으나, 현재 가장 많이 쓰고 있는 Redis, Memcached를 주로 언급한다.) 해당 분산 캐시 시스템은 대부분의 프로젝트/서비스에서 잘 활용해서 사용하고 있는 중이다.

4) 저자가 분류를 위해서 사용한 단어로 '분산 캐시 DB'라 사용한다.

그림 8_분산 캐시 시스템을 활용한 아키텍처

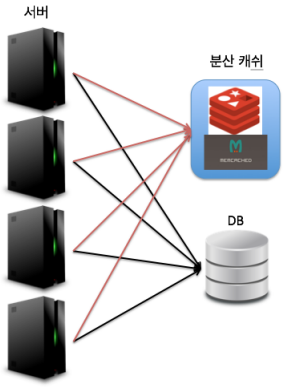


그림 9_분산 캐시 시스템 Use Case #1

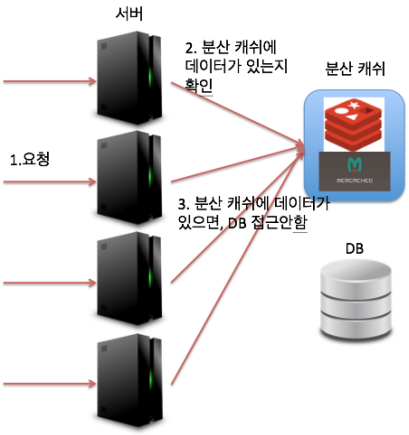
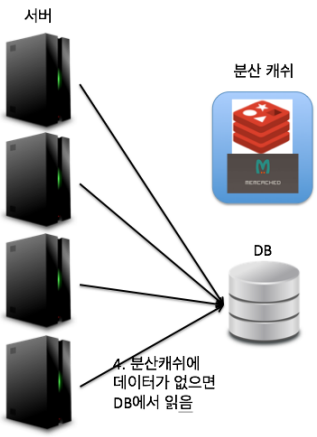


그림 10_분산 캐시 시스템 Use Case #2



〈그림 9〉와 〈그림 10〉은 분산 캐시 시스템을 이용한 Use Case이다. 서버 애플리케이션

으로 사용자 요청이 들어오면 분산 캐시에 데이터가 있는지 먼저 확인한다. 만약 캐시에 요청 데이터가 존재하면 서버 애플리케이션은 DB로 요청하지 않고 바로 사용자 요청에 대한 응답을 만든다. 하지만 분산 캐시에 데이터가 없다면 바로 DB로 요청하여 데이터를 받아 사용자 요청에 대한 응답을 만든다. (분산 캐시 내부는 1ms 이하로 응답 속도를 보내기 때문에 성능적인 저하는 크게 없는 편이다.)

메모리의 크기가 64GB인 서버에 분산 캐시 서버 10대를 설치하면 캐시 640GB의 캐시 용량을 가지고 있는 것이다. 또한 대부분 1밀리초(millisecond)의 응답속도로 캐시를 전달할 수 있다. 그러나 DB와 다르게 분산 캐시 서버의 데이터는 메모리에 저장하기 때문에 재부팅 하면 사라지는 부분이 존재한다. 그리고 메모리의 크기 이상으로 저장하지 않도록 TTL(Time to live, 캐시 데이터가 얼마나 지속할 지를 초 단위로 정함)을 명확하게 지정할 필요가 있다. 많은 개발자에게 오픈소스 커뮤니티의 참여가 가능하고, Devops 활동이 활성화되면서 여러 분산 서버 캐시 시스템 중 대표적으로 memcached와 redis가 가장 참여가 많고 안정화가 되었다. 두 분산 캐시는 공통적인 특징도 있지만 각 시스템만의 고유한 특징이 있기 때문에 설명을 진행할 예정이다.

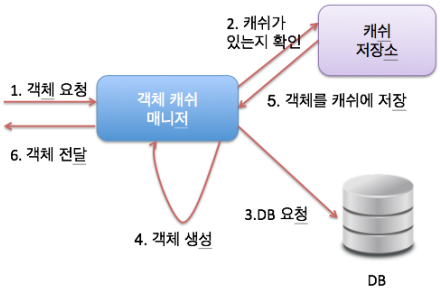
Ⅲ. 분산 캐시 사용 전의 캐시 아키텍처

Ⅱ장, '서버 환경의 캐시와 역사' 에서 언급한 분산 캐시를 사용하기 전 시대의 모델을 얘기하고자 한다. 과거의 얘기지만, 분산 캐시가 없는 환경에서도 지금도 활용할 수 있는 내용이다. 다양한 관점으로 아키텍처를 살펴볼 수 있는 기회가 될 것이다.

3.1 애플리케이션 내부 캐시

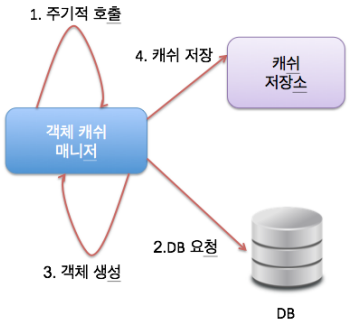
애플리케이션이 관리하는 내부 메모리를 활용한 캐시에 대해서 설명한다. 이미 위에서 내부 메모리를 활용한 캐시를 '객체 캐시'라 불렀다. 객체 캐시는 일반적으로 〈그림 10〉처럼 객체 요청을 DB로부터 요청해서 캐시에 저장하는 방식을 일반적으로 개발하거나 오픈 소스를 활용할 수 있다. 한 번 캐시 저장소에 캐시 키와 값을 저장하면, 동일한 키에 대해서 객체 캐시 매니저가 DB 요청 없이 캐시 저장소의 캐시 데이터를 전달한다.

그림 10_DB의 정보를 객체 캐시를 내부 메모리로 저장하는 방식



객체 요청에 대한 캐시 저장 뿐 아니라, 〈그림 11〉처럼 주기적인 호출(시간별, cron 표현)로 캐시를 저장하는 방식도 있다. 〈그림 10〉과 〈그림 11〉의 방식을 구현하거나 오픈 소스로 공개된 라이브러리를 사용할 수 있다.

그림 11_주기적인 호출을 통해 객체 캐시를 저장하는 방식



객체 캐시 저장소는 단순한 Map으로 구현할 수 없다. 다음의 정책이 필요하다.

- 최대 개수 제한
- 저장소 (메모리, 파일 DB) 결정
- 캐시 저장소에서 캐시가 사라지는 정책 (Garbage Collection 영향 , 개수 제한, TTL, LRU 등과 같은 알고리즘)
- 통계 또는 모니터링
- 구현 여부 (직접 구현, 오픈 소스 활용)

캐시 저장소를 직접 구현할 수 있으며, 언어 별로 많은 오픈 소스 라이브러리 등이 있다. Java의 객체 캐시 오픈 소스는 Ehcache가 가장 잘 만들어져 있다. 객체 캐시 저장소의 정책을 다 포함하고 있는 툴이라서 많은 개발자들이 애용하고 있는 오픈 소스이다.

3.2 DB (Mysql)을 이용한 캐시

현재는 잘 사용하지 않는 형태이지만, 분산 캐시 시스템을 운영하기 어려운 상황에서 쓰일 수 있다. DBA 의 도움을 받을 수 있다는 특징이 있고, 데이터의 특징이 휘발성이 아니라는 점에서 활용할 수 있는 부분이 있다. 속도가 빠르지는 않지만 미리 저장된 값을 얻을 수 있다는 점에서 활용도가 높다. DB에 '키-값' 또는 '키-맵(Map)'을 저장할 수 있는데, 〈그림 12〉와 같이 SQL 문을 정의할 수 있다. 특히 blob의 경우는 List〈Map〉 또는 Map으로 저장할 수 있는 형태인 blob 타입을 쓰고 있다.

그림 12_cache 정의 statement

```
create table simple_cache (
cache_key varchar(10) not null,
cache_value varchar(50),
update_time datetime,
primary key (cache_key)
)

// 또는

create table map_cache (
cache_key varchar(10) not null,
cache_value blob,
update_time datetime,
primary key (cache_key)
)
```

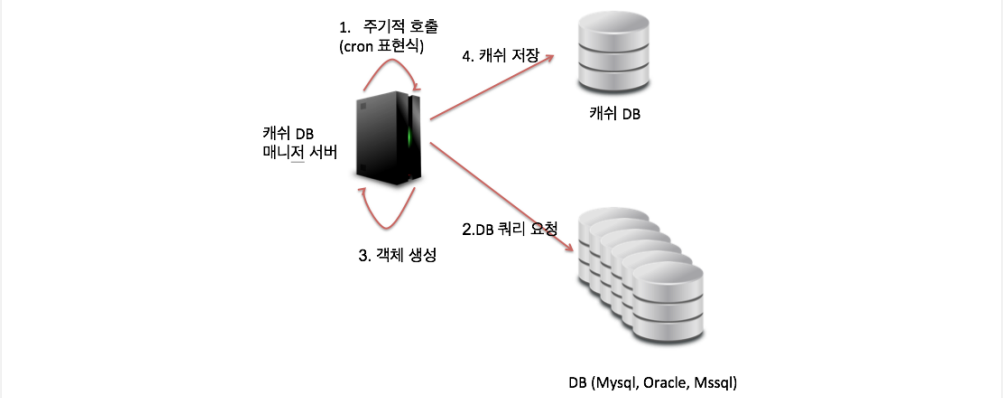
분산 캐시와 같이 DB에 저장하는 방식과 DB 데이터를 주기별로 또는 Cron 표현식 마다 DB에 SQL문을 요청하여 저장할 값을 DB에 저장하는 방식으로 나눌 수 있다. 분산 캐시와 같이 DB에 저장하는 방식은 〈그림 7〉과 같은 방식이다. 분산 캐시 시스템은 아니지만 DB에 키-값을 저장함으로서 빠른 데이터 접근을 가능하도록 할 수 있다. DB 데이터를 주기별로 요청하는 방식은 〈그림 11〉과 비슷하며, 애플리케이션 내 객체 캐시 저장 방식을 서버로 확장한 아키텍처라 할 수 있다. '캐시 DB 매니저'는 서비스에서 사용하는 DB의 정보를 주기적으로 호출하여 객체를 생성하고 캐시를 캐시 DB로 키-값 또는 키-맵으로 저장하도록 한다. 〈그림 13〉은 캐시 DB 매니저가 가지고 있는 DB 정보를 의미한다.

그림 13_캐시 DB 매니저에서 주기적으로 호출하는 서비스 DB 목록

일련번호	원본 DB	계정	SQL	cron	Update time
1	order	social	SELECT id FROM ...	0 */1 * * *	2015/1/13 17:00:00
2	profile	social	SELECT id FROM ...	0 */1 * * *	2015/1/13 17:00:00
3	delivery	social	SELECT id FROM ...	0 9 0 * * *	2015/1/13 09:00:00
4	profile	social	SELECT id FROM ...	0 10 0 * * *	2015/1/12 00:10:00
5	order	social	SELECT id FROM ...	0 10 0 * * *	2015/1/12 00:10:00
6	order	social	SELECT id FROM ...	0 0 0 * * *	2015/1/12 00:00:00
..					

〈그림 14〉는 캐시 DB 매니저와 캐시 DB/서비스 DB 간의 아키텍처를 설명하고 있다. 〈그림 13〉 과 같이 cron 표현식에 해당되는 시간이 되면, 서비스 DB에 접속하여 객체 캐시를 생성하고 캐시 DB에 캐시를 저장한다.

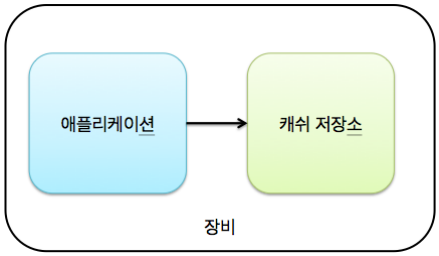
그림 14_주기적으로 호출하는 DB 캐시 아키텍처



3.3 애플리케이션 외부의 로컬 캐시

서버 애플리케이션에 2.1에서 정의한 객체 캐시 모델을 사용하는 경우, 캐시의 양이 늘어나면 메모리 사용에 제약을 받거나 Gabage Collection이 진행되면서 오랫동안 멈춰져 있는 경우(Stop the world)가 발생할 수 있다. 이를 위해서 객체 캐시를 단일 프로세스로 처리하지 않고 다른 프로세스로 두어, 메모리 사용의 오버헤드를 줄일 수 있는 경우가 있다. 한 장비에 애플리케이션 데몬과 캐시 데몬 이렇게 두 개를 두는 방식을 사용할 수 있다. 두 가지 장점이 있는데, 애플리케이션 재시작 시 캐시와 관련 정보를 바로 사용할 수 있는 부분, 캐시 메모리의 압박을 애플리케이션에서 처리하지 않아도 되기 때문에 성능은 더 높아질 수 있는 부분이 있다. 그러나 전체 아키텍처 상 캐시 데이터의 중복은 여전히 존재한다.

그림 15_한 장비에 애플리케이션과 캐시 저장소가 나누어진 채로 동작하는 애플리케이션 외부의 로컬 캐시



Ⅳ. 결론

CPU 캐시 정보를 살펴보면서 캐시의 의미를 살펴보았다. 그리고 분산 캐시 시스템은 CPU 캐시와 비슷하다. 스토리지 아닌 메모리에 저장하는 휘발성 데이터를 지원하기 때문에 빠른 접근이 가능하다. DB 응답 속도보다 더 빨리 응답하고 병렬 처리를 지원하여 애플리케이션 응답 시간을 단축한다. 또한, 여러 서버로 캐시 정보를 분산화 할 수 있고, 장애 시 중단 없이 사용할 수 있는 가용성을 지니고 있다. 그래서 사용자 증가로 인한 캐시 증가 시 동적 확장을 빨리 할 수 없다.

이런 분산 캐시 시스템이 쓰이기까지, 객체 캐시부터 분산 캐시까지의 아키텍처 역사와 아키텍처를 살펴보았다. 대용량을 지원하지 않더라도 여전히 지금도 활용 가능한 구조, 캐시 사용 시 염두 할 부분을 정리했다. Redis와 Memcached와 같은 유명한 분산 캐시 시스템을 Part 2에 이어서 설명하고자 한다.